VSIPL++: Parallel Performance

Mark Mitchell
CodeSourcery, LLC
mark@codesourcery.com

Jeffrey D. Oldham
CodeSourcery, LLC
oldham@codesourcery.com

May 27, 2004

1 Introduction

VSIPL++[1] is the object-oriented "next-generation" version of the Vector Signal and Image Processing Library (VSIPL).[2] Like VISPL, VSIPL++ specifies an Application Programming Interface (API) for use in the development of high-performance numerical applications, with a particular focus on embedded real-time systems performing signal processing and image processing. VSIPL++ contains a number of improvements relative to VSIPL including a simpler, more intuitive programming model, simpler syntax, and greater flexibility and extensibility. The most significant of VSIPL++'s improvements is its support for multi-processor computation. This parallel support requires only that the user specify the way in which data should be distributed across processors. The VSIPL++ library automatically manages the transmission of data between the processors as necessary to effectively perform the desired computations.

CodeSourcery was awarded funding under the Air Force Small Business Investment Research (SBIR) program to develop a prototype version of the parallel functionality described in the VSIPL++ specification and to obtain measurements of VSIPL++ performance when executing on parallel systems.[3] Our prototype implementation achieves a near-linear speedup on multi-processor systems demonstrating that, despite the high level of abstraction present in VSIPL++, it is nevertheless possible to obtain excellent performance. Thus, VSIPL++ has the potential to allow programmers to easily and rapidly develop systems that are both highly portable and highly efficient. In our presentation, we will describe the parallel VSIPL++ programming model, our parallel performance benchmark, and the results we obtained.

2 Benchmark Description

Beamforming is the detection of energy propagating in a particular direction while rejecting energy propagating in other directions. A beamformer consists of an array of sensors capturing signals and a signal processing algorithm to extract signals from one or more particular directions and one or more particular frequencies. The k-Ω beamformer we consider assumes uniform spacing of omnidirectional individual sensors along the x-axis. No assumptions about the signal's structure are made except that the signal is periodic and that the signal source is sufficiently far away that the signal appears planar to the sensors, and noise is assumed to be uniformly distributed across the signal.

The beamformer computes the power of the incoming signal for various bearings (k) and frequencies (Ω). Those k-Ω pairs where the power is strongest indicate incoming signals. Each sensor samples the input signal over time. After enough samples have been obtained, a computation is performed (involving the inputs from all of the sensors) to determine the power for the k-Ω pairs. First, FIR filters remove higher-order frequencies from the signal matrix. Then a real-to-complex FFT is applied to the rows of the matrix, optionally the data is reordered into a column-major matrix, and finally a complex-to-complex FFT is applied to the columns. Generally, the collection of data and determining of power is repeated multiple times. The final power reported for a given k-Ω pair is the average of that computed for the various iterations of the process.

[1] `http://www.hpec-si.org/private/vsipl++specification.html`
[2] `http://www.vsipl.org`
[3] SBIR Contract FA87450-04-C-0017

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
01 FEB 2005	**N/A**	**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
VSIPL++: Parallel Performance	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **CodeSourcery, LLC**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
See also ADM00001742, HPEC-7 Volume 1, Proceedings of the Eighth Annual High Performance Embedded Computing (HPEC) Workshops, 28-30 September 2004 Volume 1., The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**26**	

3 Implementation

We implemented the beamformer using several different programming methodologies. One implementation was written using C and VSIPL. After that implementation was complete, we developed a C++ and VSIPL++ implementation. The VSIPL and VSIPL++ implementations are similar in structure, but the VSIPL++ implementation is shorter than the VSIPL implementation because of the higher levels of abstraction provided by VSIPL++. Each implementation runs the k-Ω beamformer multiple times and computes a "running average" power spectra.

We modified the VSIPL++ reference implementation, developed by CodeSourcery under contract from MIT Lincoln Laboratory, to contain support for a subset of the functionality being considered for the parallel VSIPL++ specification. In particular, we created a data storage abstraction called `DistributedBlock` to represent a single one- or two-dimensional array that is stored across multiple processors. We specialized VSIPL++ algorithms, e.g., computing FIR filters and FFTs, to perform only local computations when operating on a `DistributedBlock`. We also modified VSIPL++ to implement a specialization of the two-dimensional FFT algorithm so that, when the input is a row-distributed `DistributedBlock` and the output is a column-distributed block, the algorithm performs the "corner-turn" required.

The VSIPL++ specification is written so as to be independent of any particular message-passing or threading system. However, in our implementation we chose to use the popular Message Passing Interface (MPI)[4] to transmit data between cooperating processors.

After making these modifications to the VSIPL++ implementation, we made minor changes to our VSIPL++ benchmark program. These changes consisted only of modifications to the types used to declare particular arrays in the benchmark program. For example, some arrays were modified to use `DistributedBlock` to indicate distribution across processors. The types of these arrays indicate the arrays are distributed by rows or columns.

4 Results

The following table demonstrates that we were able to obtain a near-linear speedup with parallel VSIPL++ relative to serial VSIPL++ and VSIPL. Times are shown for the VSIPL implementation of the benchmark, the serial VSIPL++ implementation, and the parallel VISPL++ implementation with one and two processors. Times for two hundred iterations of one problem instance are presented. Times for other instances are similar and will be presented in the extended version of this paper. In all cases, the times were obtained by running on a dual-processor Intel Pentium 4 Xeon GNU/Linux machine. The times given reflect only time spent in the execution of the beamforming computations. They do not include time required for initialization and finalization of the application and its libraries. All times shown are "wall-clock time," i.e., the total number of seconds required to execute the beamformer including time spent in the operating system kernel.

	serial, no corner turn		distributed VSIPL++	
	VSIPL	VSIPL++	1-processor	2-processor
FIR filter	20.7	20.7 + 0.1	20.7 + 0.3	20.7/2 + 0.2
1st FFT	12.7	12.7 + 0.1	12.7 + 0.0	12.7/2 + 0.3
Corner Turn	—	—	6.2 + 2.9	6.0 + 8.9
2nd FFT	10.0 + 9.2	10.0 + 9.1	10.0	10.0/2 + 0.1

The corner-turn times indicate the seconds required to transpose a row-major matrix to a column-major matrix plus the time for an MPI All-to-All computation. With one processor, only the transpose occurs. With two processors, MPI communication also occurs. The serial implementations do not perform the transpositions, leading to an increase in the time for the second FFT.

[4] `http://www-unix.mcs.anl.gov/mpi/`

Abstract Submission Details

1. Title: VSIPL++: Parallel Performance

2. Authors:

   - Mr. Mark Mitchell
     CodeSourcery, LLC
     9978 Granite Point Ct
     Granite Bay, CA 95746
     +1.916.791.8304 (voice)
     +1.916.914.2066 (fax)
     `mark@codesourcery.com`
     USA citizenship
   - Dr. Jeffrey D. Oldham
     CodeSourcery, LLC
     144 Wyandotte Dr
     San Jose, CA 95123-3727
     +1.408.578.5684 (voice)
     +1.408.578.5684 (fax)
     `oldham@codesourcery.com`
     USA citizenship

3. First Author: Mitchell
   Corresponding Author: Oldham
   Presenting Author: Oldham

4. Submit for any session.

5. Prefer talk, not poster.

6. Work areas:

   - Middleware Libraries and Application Programming Interfaces
   - Software Architectures, Reusability, Scalability, and Standards

VSIPL++: Parallel Performance

HPEC 2004

CodeSourcery, LLC

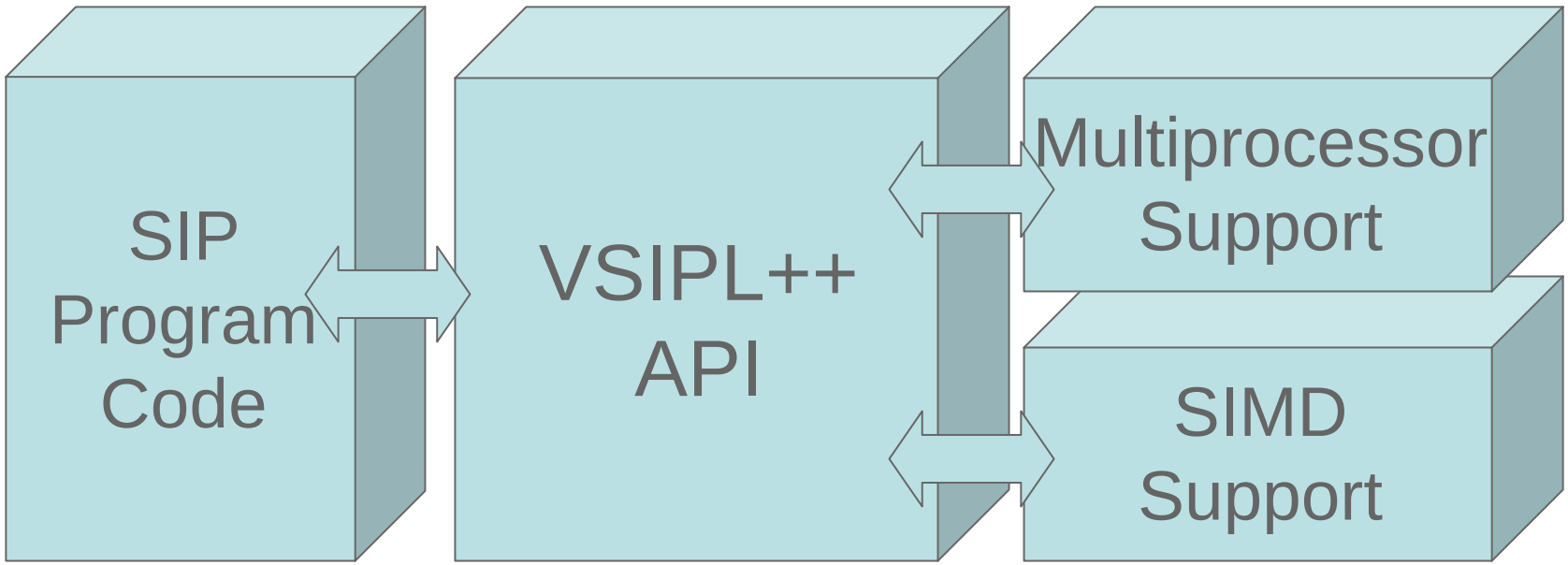
September 30, 2004

Challenge

- “Object oriented technology reduces software cost.” 

- “Fully utilizing HPEC systems for SIP applications requires managing operations at the lowest possible level.”
- “There is great concern that these two approaches may be fundamentally at odds.”

Parallel Performance Vision



"Drastically reduce the performance penalties associated with deploying object-oriented software on high performance parallel embedded systems."

"Automated to reduce implementation cost."

CodeSourcery LLC

Advantages of VSIPL

- Portability
  - Code can be reused on any system for which a VSIPL implementation is available.
- Performance
  - Vendor-optimized implementations perform better than most handwritten code.
- Productivity
  - Reduces SLOC count.
  - Code is easier to read.
  - Skills learned on one project are applicable to others.
  - Eliminates use of assembly code.

Limitations of VSIPL

- Uses C Programming Language
  - "Modern object oriented languages (e.g., C++) have consistently reduced the development time of software projects."
  - Manual memory management.
  - Cumbersome syntax.
- Inflexible
  - Abstractions prevent users from adding new high-performance functionality.
  - No provisions for loop fusion.
  - No way to avoid unnecessary block copies.
- Not Scalable
  - No support for MPI or threads.
  - SIMD support must be entirely coded by vendor; user cannot take advantage of SIMD directly.

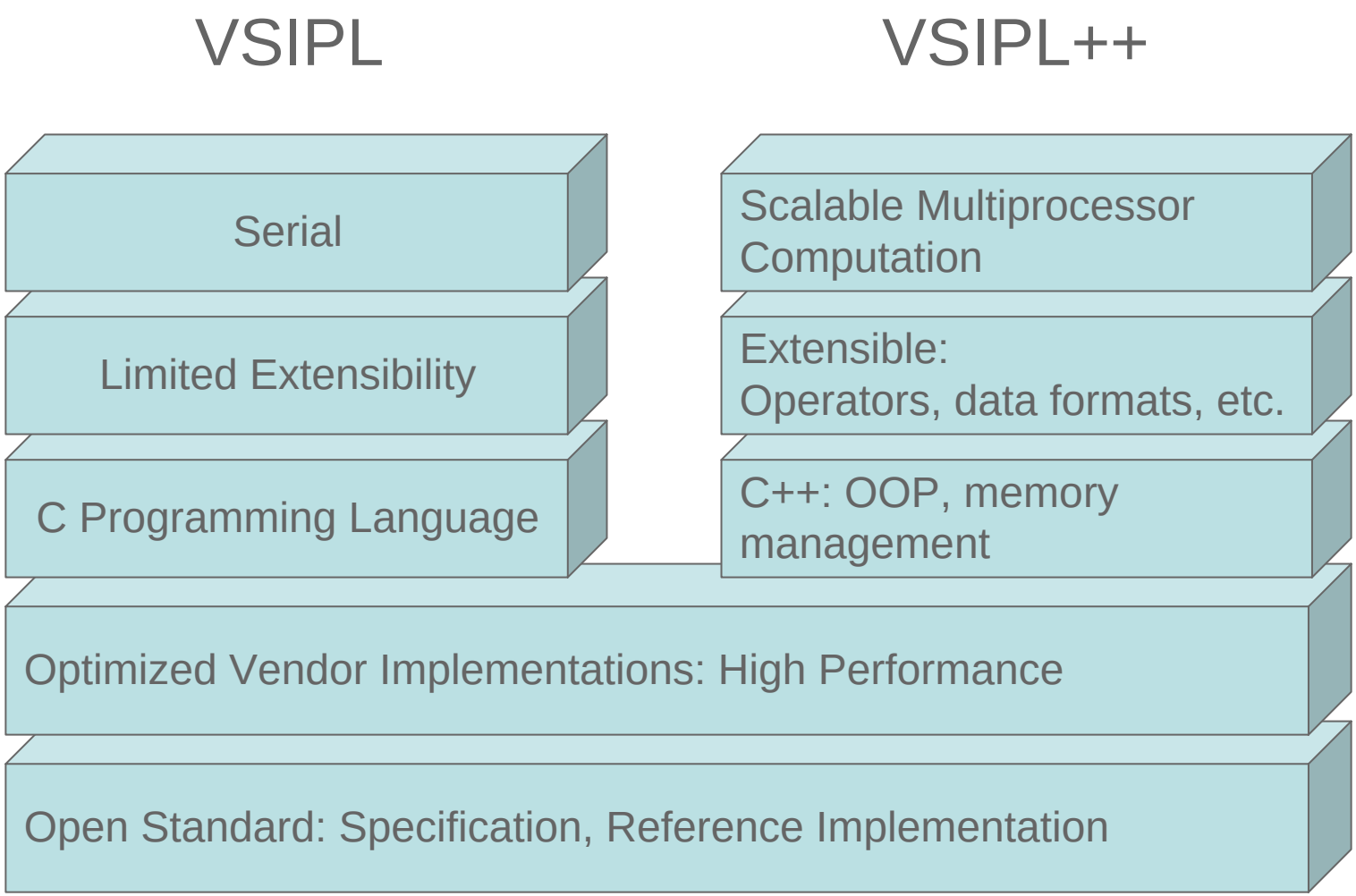
Parallelism: Current Practice

MPI used for communication, but:

- MPI code often a significant fraction of total program code.
- MPI code notoriously hard to debug.
- Tendency to hard-code number of processors, data sizes, etc.
  - Reduces portability!

- Conclusion: users should specify only data layout.

Atop VSIPL's Foundation



CodeSourcery LLC

Leverage VSIPL Model

- Same terminology:
  - Blocks store data.
  - Views provide access to data.
  - Etc.
- Same basic functionality:
  - Element-wise operations.
  - Signal processing.
  - Linear algebra.

VSIPL++ Status

- Serial Specification: Version 1.0a
  - Support for all functionality of VSIPL.
  - Flexible block abstraction permits varying data storage formats.
  - Specification permits loop fusion, efficient use of storage.
  - Automated memory management.
- Reference Implementation: Version 0.95
  - Support for functionality in the specification.
  - Used in several demo programs — see next talks.
  - Built atop VSIPL reference implementation for maximum portability.
- Parallel Specification: Version 0.5
  - High-level design complete.

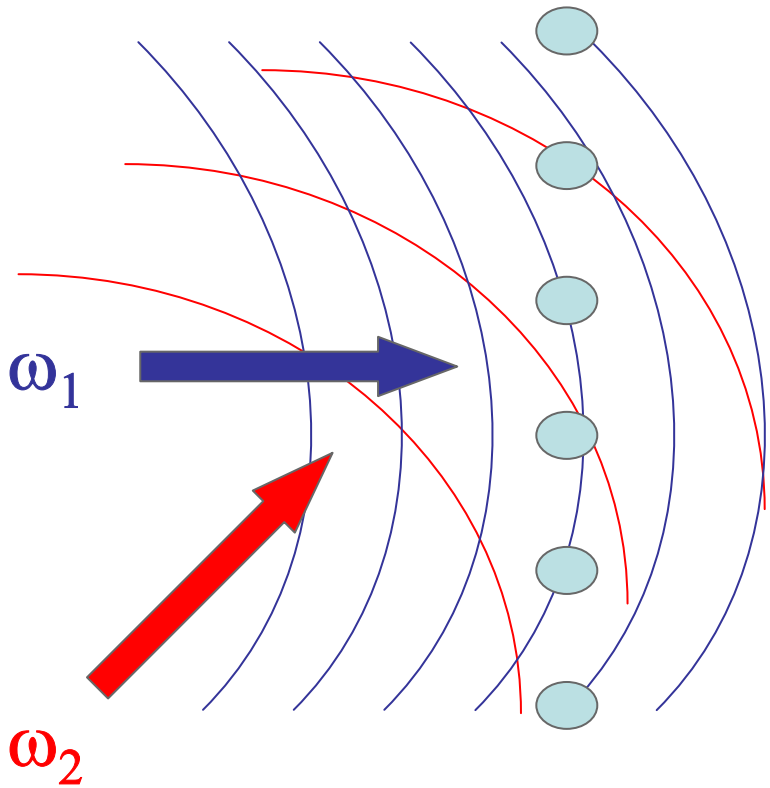
k-Ω Beamformer

Input:

- Noisy signal arriving at a row of uniformly distributed sensors.

Output:

- Bearing and frequency of signal sources.



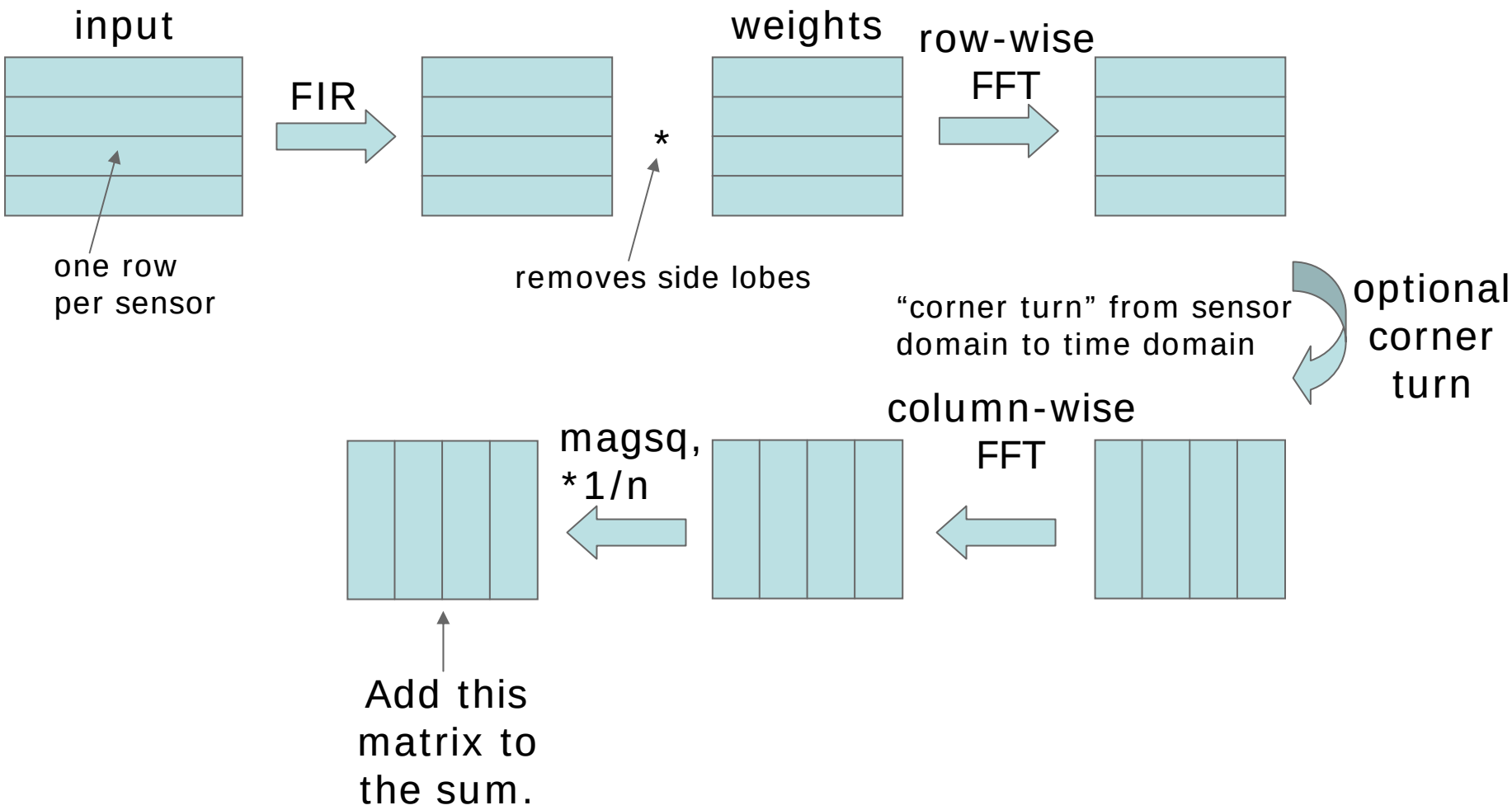
SIP Primitives Used

- Computation:
  - FIR filters
  - Element-wise operations (e.g, magsq)
  - FFTs
  - Minimum/average values
- Communication:
  - Corner-turn
    - All-to-all communication
  - Minimum/average values
    - Gather

Computation

1. Filter signal to remove high-frequency noise. (FIR)
2. Remove side-lobes resulting from discretization of data. (mult)
3. Apply Fourier transform in time domain. (FFT)
4. Apply Fourier transform in space domain. (FFT)
5. Compute power spectra. (mult, magsq)

Diagram of the Kernel



CodeSourcery LLC

VSIPL Kernel

Seven statements required:

```
for (i = n; i > 0; --i) {
  filtered = filter (firs, signal);
  vsip_mmul_f (weights, filtered, filtered);
  vsip_rcfftmpop_f (space_fft, filtered,
                    fft_output);
  vsip_ccfftmpi_f (time_fft, fft_output);
  vsip_mcmagsq_f (fft_output, power);
  vsip_ssmul_f (1.0 / n, power);
  vsip_madd_f (power, spectra, spectra);
}
```

VSIPL++ Kernel

One statement required:

```
for (i = n; i > 0; --i)
  spectra += 1/n *
    magsq (
     time_fft (space_fft (weights *
                          filter (firs,
                                  signal)));
```

No changes are required for distributed operation.

Distribution in User Code

Serial case:

```
Matrix<float_t, Dense<2, float_t> >
   signal_matrix;
```

Parallel case:

```
typedef Dense<2, float_t> subblock;
typedef Distributed<2, float_t, subblock, ROW>
  Block2R_t;
Matrix<float_t, Block2R_t> signal_matrix;
```

User writes no MPI code.

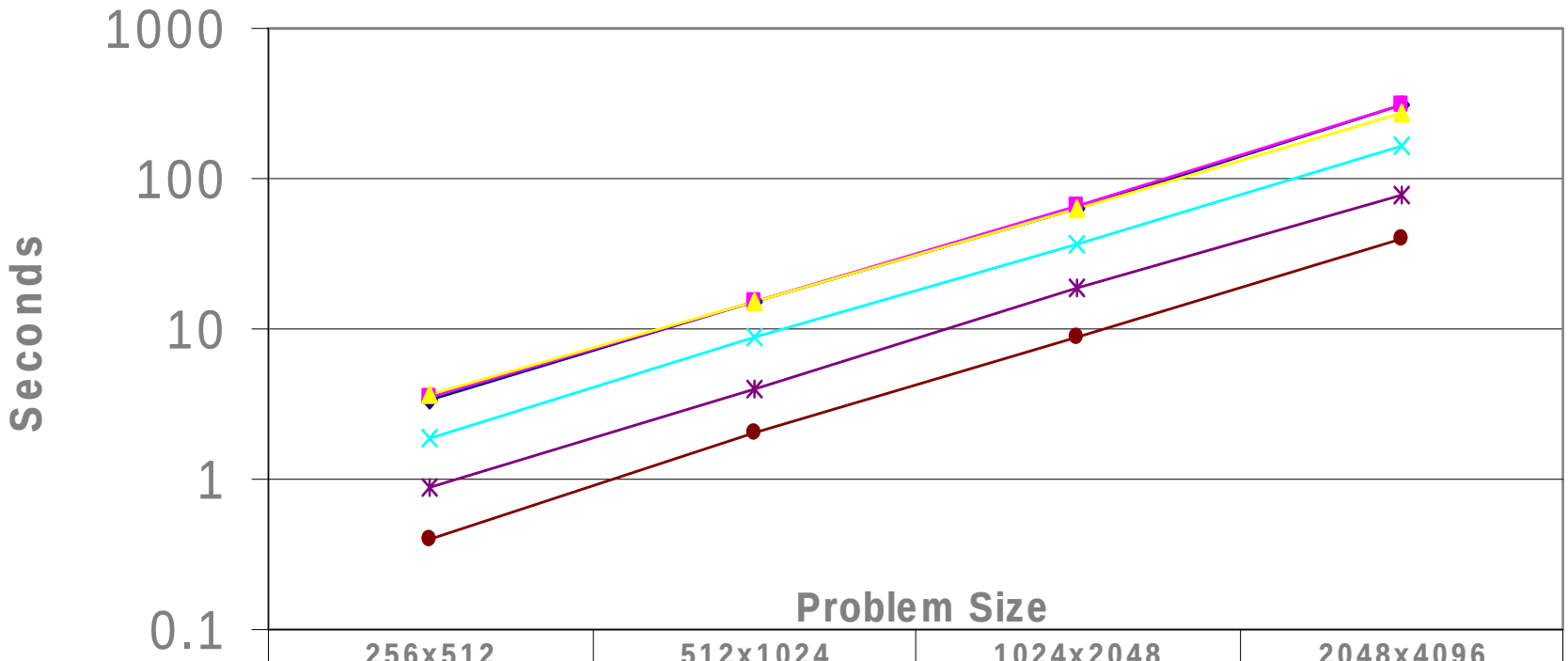
VSIPL++ Implementation

- Added `DistributedBlock`:
  - Uses a "standard" VSIPL++ block on each processor.
  - Uses MPI routines for communication when performing block assignment.
- Added specializations:
  - FFT, FIR, etc. modified to handle `DistributedBlock`.

Performance Measurement

- Test system:
  - AFRL HPC system
  - 2.2GHz Pentium 4 cluster
- Measured only main loop
  - No input/output
- Used Pentium Timestamp Counter
- MPI All-to-all not included in timings
  - Accounts for 10-25%

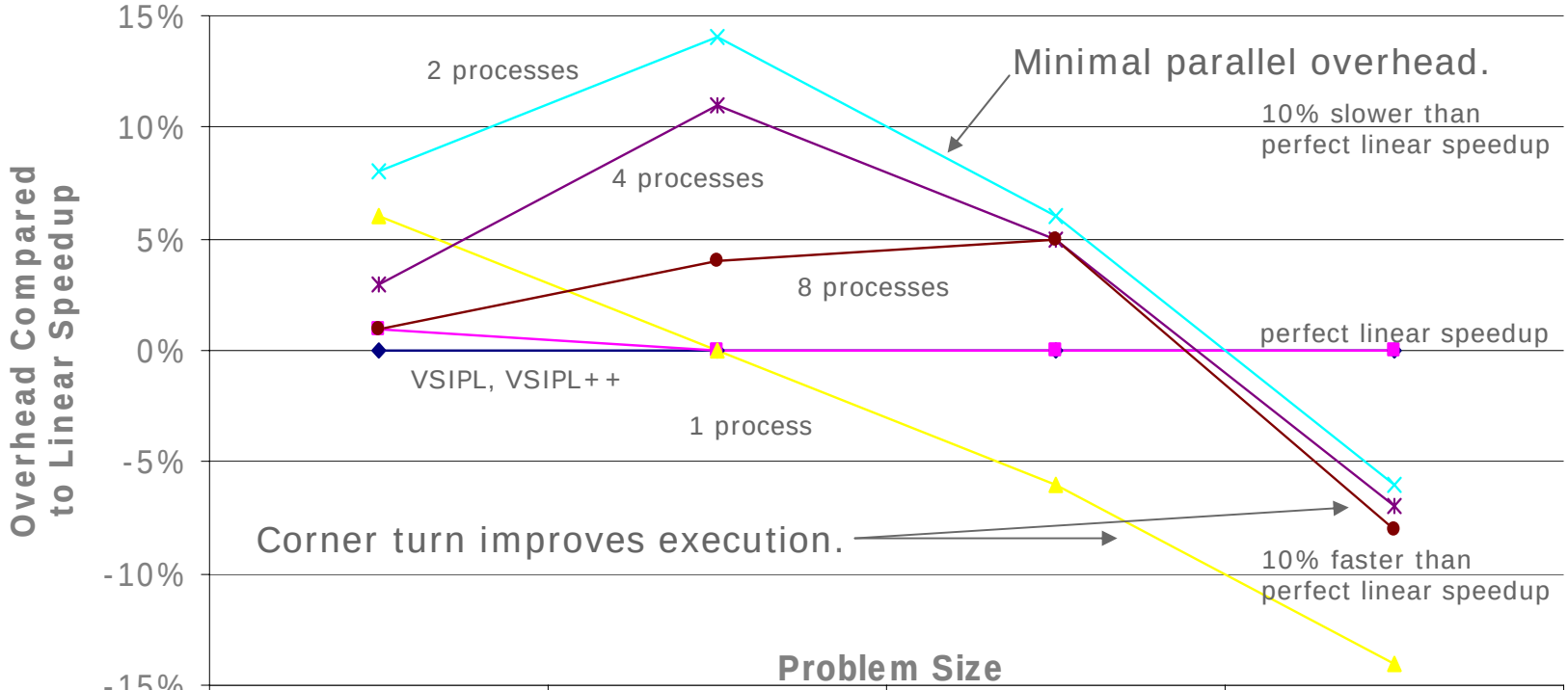
VSIPL++ Performance



	256x512	512x1024	1024x2048	2048x4096
VSIPL	3.3	15	64	306
VSIPL++	3.5	15	66	314
|| VSIPL++ (1)	3.6	15	63	277
|| VSIPL++ (2)	1.9	9	37	165
|| VSIPL++ (4)	0.9	4	19	78
|| VSIPL++ (8)	0.4	2	9	40

CodeSourcery LLC

Parallel Speedup



	256x512	512x1024	1024x2048	2048x4096
VSIPL	0%	0%	0%	0%
VSIPL++	1%	0%	0%	0%
|| VSIPL++ (1)	6%	0%	-6%	-14%
|| VSIPL++ (2)	8%	14%	6%	-6%
|| VSIPL++ (4)	3%	11%	5%	-7%
|| VSIPL++ (8)	1%	4%	5%	-8%

CodeSourcery LLC

Conclusions

- VSIPL++ imposes no overhead:
  - VSIPL++ performance nearly identical to VSIPL performance.
- VSIPL++ achieves near-linear parallel speedup:
  - No tuning of MPI, VSIPL++, or application code.
- Absolute performance limited by VSIPL implementation, MPI implementation, compiler.

VSIPL++

Visit the HPEC-SI website

http://www.hpec-si.org

- for VSIPL++ specifications
- for VSIPL++ reference implementation
- to participate in VSIPL++ development

VSIPL++: Parallel Performance

HPEC 2004

CodeSourcery, LLC

September 30, 2004